SmartModels : la généricité paramétrée au service des modèles métiers

Pierre Crescenzo, Philippe Lahire, Emanuel Ţundrea

SmartModels : la généricité paramétrée au service des modèles métiers

Pierre Crescenzo* — **Philippe Lahire*** — **Emanuel Ţundrea****

** Laboratoire I3S (UNS/CNRS) - Projet OCL*
2000 route des lucioles, Les Algorithmes, bâtiment Euclide - BP 121
F-06903 Sophia-Antipolis cedex - France
Pierre.Crescenzo@unice.fr, Philippe.Lahire@unice.fr

*** Faculty of Automatics and Computer Science*
"Politehnica" University of Timişoara - Bd. V. Parvan no 2, 1900 Timişoara - Romania
Emanuel@emanuel.ro

RÉSUMÉ. *Il est devenu vital de pouvoir faire évoluer la structure des entités logicielles, de développer rapidement de nouvelles fonctionnalités et de prendre en compte de nouvelles contraintes du contexte d'exécution. L'approche proposée pour atteindre cet objectif s'intègre dans le contexte de l'ingénierie des modèles. Nous proposons un métamodèle pour décrire des modèles métiers et un mécanisme pour les composer. Une des originalités de notre métamodèle repose sur le fait que le concepteur du modèle métier peut s'appuyer sur la description d'entités génériques dont le degré de généricité est défini au moment de la conception du modèle. Nous centrons la présentation sur cet aspect et montrons un exemple de ligne de produits d'appareils photographiques. La validation de notre approche est faite par une extension d'EMF sur Eclipse.*

ABSTRACT. *It is essential to be able to make evolutions on the structure of software entities, to quickly develop new functionalities and to take into account new constraints of the execution context. The approach we propose to reach this goal is integrated in the context of model engineering. We propose a metamodel to describe business models and a mechanism to compose them. One of the originalities of our metamodel is that the designer of the business model can use descriptions of generic entities with a genericity degree which is defined during the model design. We focus our presentation on this aspect and use an example of a product line of cameras. The validation of our approach is made by an extension of EMF in Eclipse.*

MOTS-CLÉS : *généricité paramétrée, modèles métiers, métamodélisation*

KEYWORDS: *Parametrised Genericity, Business Models, Metamodelling*

1. Introduction

Les nécessités d'évolution des logiciels orientés objets impliquent de pouvoir adapter l'existant pour le réutiliser : réutilisation de classes ou de bibliothèques de classes, du modèle de l'application ou bien encore du savoir-faire. De nombreux paradigmes sont apparus pour répondre à cette problématique. Nous pouvons citer, par exemple, la séparation des préoccupations, la généricité, les approches dirigées par les modèles, la métamodélisation ou la programmation par composants. L'approche que nous proposons dans cet article s'appuie sur plusieurs de ces paradigmes en vue de mener à une solution pour la réutilisation des modèles d'applications.

La spécification de modèles, notamment métiers, implique la prévision de différentes variantes d'une même entité pour définir des lignes de produits. En d'autres termes, nous pensons que certaines entités d'un modèle métier sont génériques et cette généricité doit permettre une variation aussi bien structurelle que comportementale des entités. C'est l'introduction de cette généricité[1] que nous souhaitons développer ici.

Par ailleurs, en s'appuyant sur les idées véhiculées par la séparation des préoccupations, notre approche encourage la définition de petits modèles (avec peu d'entités) : chacun décrit une part limitée de la problématique correspondant à la ligne de produits ou plus généralement au modèle métier. C'est un avantage parce que cela permet de bien localiser chaque entité et fonctionnalité et d'augmenter la réutilisabilité. Et nous défendons l'idée que cela n'augmente pas pour autant les difficultés lors de la composition de ces modèles (conflits, renommages, adaptations...). En effet, notre démarche inclut un protocole de composition qui simplifie et automatise nettement cette tâche. Il n'est pas l'objet essentiel de cet article mais il constitue un aspect important de notre approche globale (Crescenzo *et al.*, 2005).

En privilégiant des petits modèles, génériques et facilement réutilisables, nous travaillons dans l'optique de simplifier leur réutilisation effective.

Dans la section 2, nous expliquons les différentes étapes qui nous ont conduits à la proposition d'un métamodèle pour la spécification de modèles métiers et nous décrivons les principaux aspects de ce métamodèle. Dans la section 3, nous montrons que notre approche, basée sur une généricité paramétrée, fournit l'expressivité nécessaire à la modélisation des lignes de produits. La section 4 décrit les principaux éléments d'un *plugin* Eclipse (Eclipse foundation, 2004) pour la mise en œuvre de notre approche. Enfin, dans les deux dernières sections, nous situons notre travail par rapport à l'état de l'art puis nous concluons en donnant des perspectives.

1. La généricité regroupe traditionnellement les problèmes de variabilité et ceux d'adaptation à un domaine ou un rôle spécifique.

2. Présentation du métamodèle *SmartModels*

Historiquement, notre intérêt pour le paramétrage générique a commencé avec (Crescenzo, 2001). L'objectif de cette thèse est la modélisation des langages à objets, grâce à un métamodèle et à un protocole métaobjet, en introduisant le concept de *paramètre hypergénérique*[2].

Ce système, formé d'un métamodèle et d'un protocole métaobjet, porte le nom d'OFL (*Open Flexible Languages*). Il offre, au travers d'entités méta et de paramètres, la possibilité de représenter, modéliser et adapter un langage à objets. Par exemple, nous y retrouvons les entités méta suivantes[3] : *métaclasse* (concept traditionnel), *métarelation* (pour modéliser le comportement des héritages et des agrégations, par exemple), *métalangage* (pour réunir des métaclasses et métarelations). Chacune de ces entités est définie au moyen de paramètres génériques. Par exemple, la métaclasse possède un paramètre booléen *créateur* qui représente la capacité de la classe à créer ou non des instances. Ainsi, en nous plaçant en Java, *Class* a une valeur *vrai* pour ce paramètre et *Interface* une valeur *faux*. Et cela conditionne leur comportement. Il nous a ensuite paru intéressant d'ajouter à OFL des caractéristiques non prévues initialement. C'est dans cette optique que nous avons entrepris des travaux sur la séparation et de la composition de préoccupations (Lahire *et al.*, 2006) et nous avons pensé que les idées et concepts issus de ces approches pouvaient être transposés avec profit dans le contexte de la modélisation des applications et, en particulier, des modèles métiers. C'est donc à ce niveau-là que nous appliquons la métamodélisation avec paramètres génériques (Crescenzo *et al.*, 2005), pour augmenter l'expressivité des modèles, et les adaptateurs pour aider à la composition de ces modèles (Lahire *et al.*, 2006).

2.1. *Le métamodèle*

Pour spécifier un modèle métier, nous nous reposons, comme souvent, sur un métamodèle qui permet notamment de définir le niveau de généricité de chaque entité. Il est ainsi possible d'aboutir à la description d'entités dérivées par simple instanciation, la partie générique offrant la variabilité. Les éléments de base nécessaires à la spécification d'un modèle métier (attributs, associations, méthodes, classes, types de base, etc.) sont présents dans les métamodèles courants tels que MOF, UML ou EMF, qui possèdent aussi la capacité de se décrire eux-mêmes. Le métamodèle *SmartModels*

2. Le préfixe *hyper* symbolise le fait que ces paramètres influent globalement sur l'ensemble du comportement du système modélisé. Avec le temps et dans une volonté de simplifier le vocabulaire, il a tendance à disparaître de nos écrits, sans pour autant que son sens se soit affaibli.

3. Leur nom a été adapté pour cet article. Attention, nous sommes ici au niveau d'un langage et non d'une application. Cela signifie, par exemple, que le mot *classe* représente la notion de classe et que la *métaclasse* est une entité capable de générer différents types de classe (et non différentes classes). En nous plaçant au niveau de l'application, elle s'appellerait *métamétaclasse*.

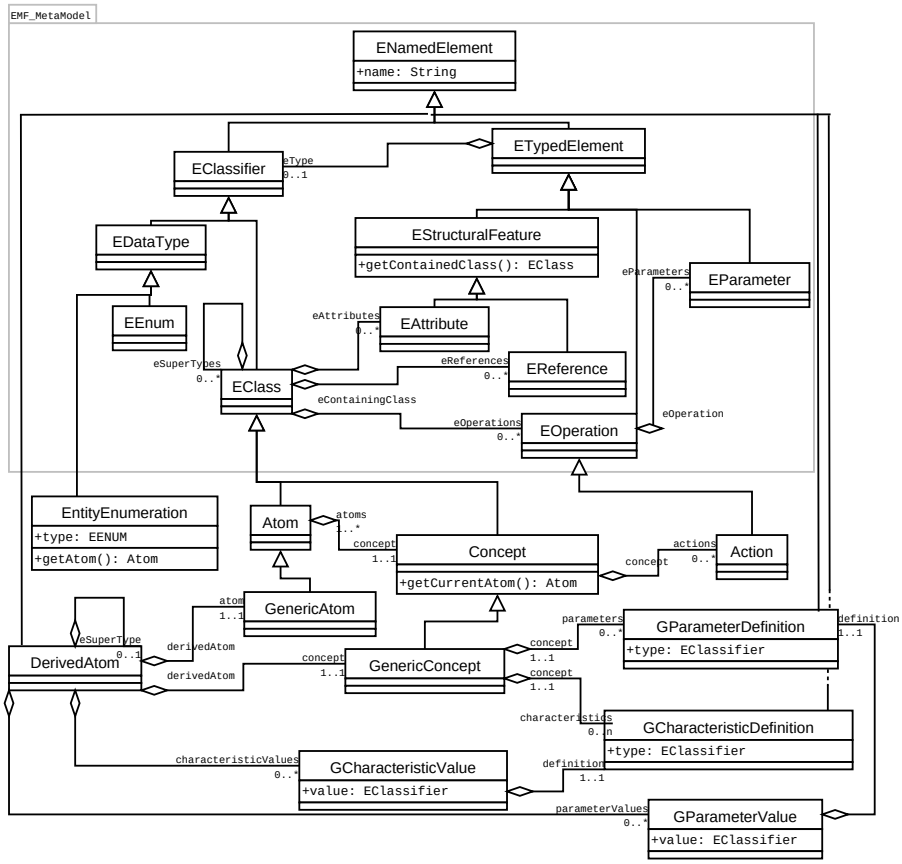

Figure 1. *Architecture simplifiée de* SmartModels

que nous présentons est une extension de EMF choisi pour deux raisons : *i)* il est à la fois réduit et suffisamment expressif pour décrire un modèle métier et *ii)* il fait partie intégrante de l'environnement Eclipse ce qui ouvre des possibilités évidentes pour la réalisation d'un prototype. Les entités de *SmartModels* et des modèles qu'il décrit, sont équipés d'assertions implémentées avec une extension d'EMF pour la description de contraintes en OCL(OMG, 2003; Vanwormhoudt, 2005).

La figure 1 présente un extrait de notre métamodèle[4] et des principales entités du modèle EMF sur lesquelles s'appuie l'extension. Les entités qui constituent le modèle EMF d'Eclipse sont présentées dans le grand rectangle *EMF_MetaModel*. Il est possible d'y mettre en évidence une racine *ENamedElement* qui garantit que chaque entité est nommée. *EClassifier* est l'entité qui représente un équivalent des classifieurs

4. Les noms des entités ont parfois été volontairement raccourcis pour économiser de l'espace.

d'UML : tout ce qui ressemble à une classe (conteneur d'objets) ou à un paquetage (conteneur de classes ou de conteneurs). *EAttribute* et *EOperation* réifient les concepts d'attribut et de méthode. Tout ce qui se trouve hors d'*EMF_MetaModel* fait partie de notre extension et est présenté dans la sous-section suivante.

2.2. *Éléments de la sémantique du métamodèle*

Les deux entités principales de notre extension sont les concepts (*Concept*) et les atomes (*Atom*). Les atomes représentent les différents types d'éléments que nous pouvons manipuler dans les modèles, instances de *SmartModels*. Les concepts sont les métaentités des atomes. Dans notre approche, les concepts décrivent entre autres les paramètres génériques et les atomes leur donnent une valeur. Les atomes dérivés (*DerivedAtom*) sont des atomes qui ne sont pas fournis par défaut par le métamodèle lui-même (ils sont donc spécifiques à un modèle ou à un ensemble de modèles). Conformément à leur définition, les atomes sont généralement abstraits et les atomes dérivés souvent concrets.

Les concepts génériques (*GenericConcept*) sont composés de paramètres (*GParameterDefinition*), mais aussi de caractéristiques (*GCharacteristicDefinition*)[5]. Pour ne pas alourdir inutilement le schéma, nous n'avons pas représenté la sous-hiérarchie de leurs types de base (ex : paramètre de type entier ou de type tuple d'entiers). Il en va de même pour les valeurs qui en sont issues (*GParameterValue* et *GCharacteristicValue*). Nous avons donc remplacé le type réel des primitives *type* et *value* par *EClassifier* ce qui signifie que le type considéré peut être une classe (y compris de collections, de tuples…) ou un type prédéfini (entier, énumération…). La hiérarchie complète réelle est de plus conçue pour être compatible avec la hiérarchie des types d'OCL. Les caractéristiques offrent la possibilité de représenter des éléments de la structure des atomes. Leur domaine de valeur est fréquemment lié, par des conditions ou des contraintes, à un ou plusieurs paramètres. Les derniers composants importants des concepts sont les actions (*Action*). Elles représentent la partie exécutable (les algorithmes) de la réification. Elles sont presque toujours dépendantes de paramètres et/ou de caractéristiques.

La figure 1 montre qu'un atome peut spécialiser un ou plusieurs autres atomes de même qu'un concept peut lui aussi hériter de plusieurs concepts. Un atome dérivé peut être construit à partir d'un autre : l'intérêt est ici de pouvoir réutiliser une partie des valeurs de paramètres et de caractéristiques, associée à un atome dérivé, et de n'avoir à redéfinir que les valeurs qui varient. Cette redéfinition est contrôlée par les règles de redéfinition associées à chaque paramètre ou caractéristique.

Nous avons vu plus haut que le concept représente la partie méta d'un atome : il règle le comportement commun à l'ensemble des instances d'un même atome, en

5. Un paramètre représente une valeur simple, généralement entière, booléenne ou énumérée dans une liste prédéfinie. Une caractéristique est plutôt une valeur multiple, souvent une liste de taille quelconque d'objets complexes.

fonction de la valeur de ses paramètres et caractéristiques. Le fonctionnement peut dépendre de la structure (propriétés, méthodes) d'un atome. Un concept a accès à l'ensemble des informations associées à la structure d'un atome et peut être associé à plusieurs atomes. Il limite généralement les hypothèses qu'il fait sur le contenu des atomes qu'il gère en ne considérant que leur partie commune. Ainsi, il est intéressant d'utiliser l'héritage multiple entre atomes pour spécifier la partie commune de plusieurs atomes et, dans certaines situations, il est opportun de calquer la hiérarchie des concepts sur celle des atomes. S'il n'est pas explicitement rattaché à un concept par le créateur du modèle, un atome est rattaché au même concept que son parent.

2.3. *Méthodologie associée*

Le concepteur d'un modèle métier doit tout d'abord identifier les différentes entités du domaine modélisé et établir le degré de généricité[6] de chaque entité. Il commence par décrire les entités qui ne sont pas génériques (c'est-à-dire celles qui ont une structure et un comportement stables dans le temps et qu'il n'est donc pas utile de rendre variables). Elles sont représentées par un atome (instance de la classe *Atom* ou d'une de ses sous-classes non héritière de *GenericAtom*). L'atome peut éventuellement ne reposer que sur des actions (correspondant ici à des méthodes de classe) si un concept non générique lui est associé : l'absence de variation rend les paramètres et caractéristiques superflus. Les atomes non génériques sont instances de *Concept* et non de *GenericConcept*

Pour les entités génériques, le concepteur définit une métaclasse générique (héritière de *GenericConcept*) et y décrit des paramètres et des caractéristiques qui vont permettre de faire varier la structure et le comportement de ces entités. Ils sont respectivement définis comme des instances des classes *GParameterDefinition* et *GCharacteristicDefinition*. Sur le même principe, les actions sont ensuite décrites. L'étape suivante consiste à écrire les contraintes OCL qui s'appliquent : préconditions, postconditions et invariants. Une fois toutes les entités créées, il reste à concrétiser le modèle en instanciant les concepts pour créer les atomes dérivés, génériques ou pas. D'un point de vue pratique, les informations relatives à l'ensemble des entités évoquées ci-dessus peuvent être saisies grâce à un éditeur graphique (voir section 4).

Une fois le modèle métier entièrement décrit, ce dernier va subir une suite de transformations pour aboutir à un ensemble de classes compilable qui forme une application ou une bibliothèque. Des outils logiciels (par exemple, des *visiteurs* sur les entités du modèle) sont également produits : ils permettent d'équiper l'application ou la bibliothèque de diverses fonctionnalités (exemple : trace, introspection, persistance...).

6. Nous entendons par *degré de généricité* le nombre de paramètres génériques de chaque entité. Ce nombre, comme le choix de ce que décrivent les paramètres, est bien entendu affaire de compromis et doit être laissé sous la responsabilité du concepteur du métamodèle.

3. Généricité pour les lignes de produits

Pour décrire un modèle métier et, en particulier, une ligne de produits (Ziadi, 2004) nous avons besoin de *i)* décrire la structure et le comportement d'un produit ainsi que les éventuelles parties communes qui peuvent exister entre deux produits, *ii)* proposer des variantes d'un même produit et donc spécifier des variations dans la structure ou dans les fonctionnalités offertes, *iii)* choisir une entité parmi plusieurs alternatives possibles et dégager éventuellement une partie commune entre ces entités, *iv)* décrire les contraintes d'assemblage et d'exécution entre les entités comme, par exemple, une contrainte d'exclusion mutuelle et, *v)* définir les variations de comportement qui découlent de la spécification des variantes d'un produit. Les différents exemples d'extrait de modèle proposés dans la suite ne contiennent bien évidemment pas tous les attributs, méthodes et associations.

3.1. *Exprimer la variation structurelle*

L'aspect structurel d'une entité métier est décrit dans un atome. Les modèles à objets comme UML ou EMF permettent de décrire un certain degré de variabilité par des associations ou héritages. La description d'un atome permet bien entendu cette variabilité puisqu'il est possible, dans un atome, de signaler à la fois un héritage vers un autre atome, mais aussi des attributs et associations ou encore l'existence de contraintes. Nous pouvons toutefois aller plus loin en exprimant le fait que des entités peuvent avoir des structures différentes et une sémantique assez proche (section 3.1.1), que des propriétés voire des entités peuvent être optionnelles (section 3.1.2) ou représenter des choix alternatifs (section 3.1.3).

3.1.1. *Faire varier le nombre de propriétés ou de méthodes*

Prenons l'exemple de la modélisation simplifiée d'un appareil photographique (argentique ou numérique). Plusieurs axes de modélisation sont possibles, par exemple celui des *points de vue* (points de vue technique, documentation ou comportement) et celui des *fonctionnalités* (fonctionnalités de prise de vue, de stockage ou de communication).

Dans cet article, nous prenons pour exemple un modèle orienté vers les fonctionnalités de communication de l'appareil photographique avec son environnement (figure 2). Dans notre exemple, cette communication s'appuie sur différents ports (USB, BlueTooth, etc.). La figure 2 ne représente que les entités utiles au développement et à la compréhension de l'exemple. Une entité (par exemple l'atome générique *AppareilPhoto*) peut, selon le type d'appareil, avoir des propriétés et des méthodes différentes.

Pour cela nous pouvons créer des descendants (*AppareilArgentique* et *AppareilNumérique*) de telle manière que le concept associé à chacun d'eux (respectivement *AppareilArgentiqueSem* et *AppareilNumériqueSem*) soit une spécialisation de *AppareilPhotoSem* (concept générique pour tous les appareils photographiques). La relation d'héritage utilisée entre les atomes est également essentielle. Signalons en effet que le

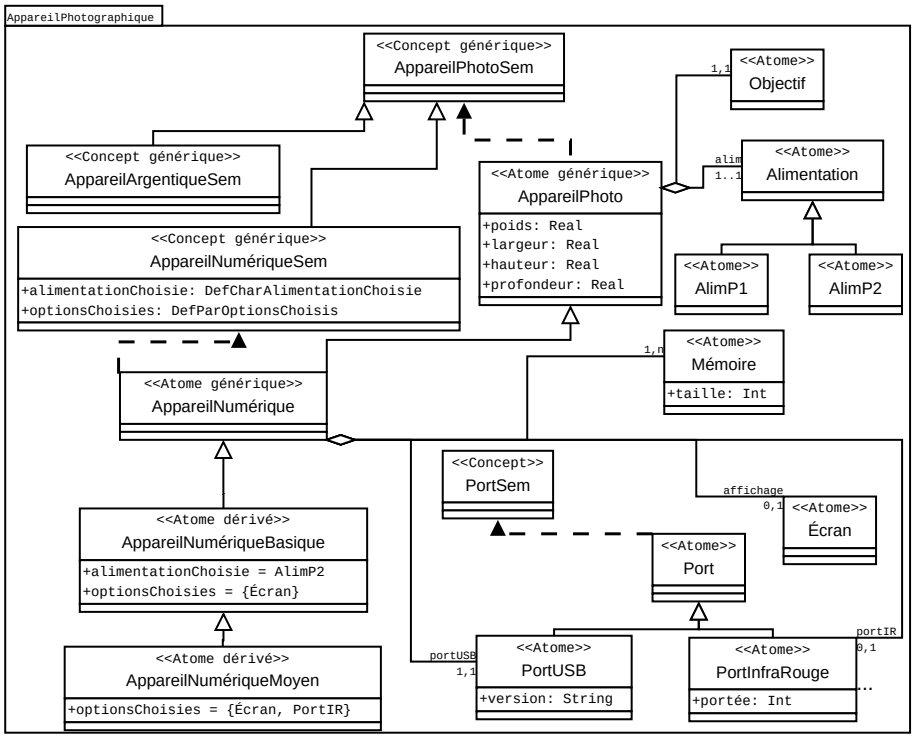

Figure 2. *Modélisation d'un appareil photographique*

fait que les trois atomes aient le même concept (ou des concepts qui se spécialisent) n'indique en rien une hiérarchie obligatoire entre eux.

Pour les entités non génériques, nous pouvons comme précédemment exploiter le fait qu'un concept est partagé ou non ou encore que deux concepts ont une racine commune. Nous pouvons aussi considérer seulement les différences structurelles des atomes sans tenir compte de leurs concepts respectifs. Nous verrons dans la section 3.1.3 comment exploiter l'information capturée par les hiérarchies de concepts et d'atomes.

3.1.2. *Définir des propriétés ou des méthodes optionnelles*

Le caractère optionnel d'une entité, d'une propriété ou d'une fonctionnalité peut être spécifié à travers la définition d'un paramètre et de contraintes. La définition du paramètre permet de spécifier la liste des propriétés optionnelles et les propriétés effectivement sélectionnées sont spécifiées lors de la construction du produit dérivé (*AppareilNumériqueMoyen* sur la figure 2). Un paramètre (ou une caractéristique) est défini par un nom, un type et un ensemble de redéfinitions possibles. La figure 3 propose une

description du paramètre *optionsChoisies* mentionné dans la figure 2. La valeur que prend un paramètre ou une caractéristique est définie au moment où l'atome dérivé est construit.

```
nom : optionsChoisies -- Définition de caractéristique pour l'assemblage
rôle : définir les propriétés optionnelles possibles d'un appareil
type : Collection d'Énumération -- = un ensemble de noms de propriété
     -- ici, l'énumération contient les options affichage et portIR
valeurs possibles : {}, {"affichage"}, {"portIR"}, {"affichage", "portIR"}
valeur par défaut : {}
redéfinition possible dans un concept plus spécialisé : libre
```

Figure 3. *Définition de propriétés optionnelles*

Un aspect important de cette approche, par rapport à l'utilisation d'un stéréotype *optional* dans un diagramme UML (Claub, 2001; Ziadi, 2004) est que même si une propriété (attribut, association…) est optionnelle, elle est déclarée dans l'atome. Cette déclaration peut paraître inutile quand l'option n'est pas sélectionnée dans le produit dérivé. Cependant il est intéressant que le jeu d'options puisse être modifié en intervenant au niveau de la définition du paramètre, que ce soit par modification directe, par redéfinition dans un descendant, ou encore par ajout dû à une composition, sans toucher à la structure même de l'atome (Crescenzo *et al.*, 2005). Cela semble d'ailleurs aller dans le sens de l'évolution naturelle de l'offre des options pour un produit (une option sur le long terme a tendance à être intégrée dans l'offre de base). La figure 4 présente la contrainte OCL associée à la définition d'optionnalité de la figure 3. Notons l'existence et l'utilisation de *getCurrentAtom* qui est une méthode de la classe *Concept* de notre modèle, qui permet de faire référence à l'atome (dérivé) associé à un concept pour une instance d'atome donné[7]. L'atome dérivé représente un type d'appareil photographique plus précis, parmi ceux qui peuvent être créés d'après la description de la ligne de produits.

```
context DefParOptionsChoisies
inv cohérenceDesOptions : -- contrainte d'assemblage
let derivedAtom : Atom = concept.getCurrentAtom()
    derivedAtom.optionsChoisies->forAll (e : EEnum |
        derivedAtom.attributes()->includes(e) or
        derivedAtom.associationEnds()->includes(e))
```

Figure 4. *Contrainte relative à la définition de propriétés optionnelles*

Une seconde approche consisterait à construire un modèle de base, sans partie optionnelle et un second modèle contenant les entités optionnelles et les parties des entités du modèle de base qui sont optionnelles. La construction du modèle complet serait alors réalisée par composition des deux modèles suivant l'approche définie dans (Crescenzo *et al.*, 2005). Cette approche est techniquement tout aussi valide mais est à privilégier, à notre avis, plutôt quand les parties optionnelles sont très limitées.

7. Détail technique : le *plugin* Eclipse qui nous permet d'utiliser des contraintes OCL offre un accès direct aux méthodes de *EClass*. Sur le même principe, nous accédons directement à *getCurrentAtom* qui est une méthode d'une descendante de *EClass*, en l'occurrence *Concept*.

3.1.3. *Définir des entités alternatives*

Le modèle décrit sur la figure 2 exprime qu'un appareil photographique numérique (*AppareilNumérique*) contient un moyen d'alimentation et un seul, en l'occurrence *AlimP1* ou *AlimP2*. Dans notre approche, les atomes décrivent *i)* les propriétés et les fonctionnalités d'instance, c'est-à-dire tout ce qui participe à la description d'un exemplaire d'appareil photographique, *ii)* les détails concernant la sémantique d'un type particulier d'appareil photographique, c'est-à-dire les contraintes et la cohérence de l'assemblage des différents composants et *iii)* le fonctionnement de l'appareil. Ces trois éléments sont regroupés dans le concept *AppareilNumériqueSem*[8] qui est attaché à l'atome. Ainsi nous définissons dans *AppareilNumériqueSem* une caractéristique *AlimentationChoisie* qui contient la valeur choisie (figure 5).

```
nom : AlimentationChoisie -- Définition de caractéristique pour l'assemblage
rôle : définir des alternatives possibles d'alimentations pour un appareil numérique
type : EntityEnumeration -- = une énumération de noms d'entités
    ici, l'énumération contient les alimentations valides : AlimP1 et AlimP2
valeurs possibles : AlimP1 ou AlimP2
valeur par défaut : AlimP1
redéfinition possible dans un concept plus spécialisé : libre
```

Figure 5. *Définition d'entités alternatives*

Dans notre cas, la contrainte sur la définition de la caractéristique *AlimentationChoisie* (figure 6) permet de garantir que l'alimentation associée à tout appareil numérique est valide.

```
context DefCharAlimentationChoisie
inv cohérenceDesAlimentations : -- contrainte d'assemblage
let derivedAtom : Atom = concept.getCurrentAtom()
    derivedAtom.alimentationChoisie.getAtom()->notEmpty and
    derivedAtom.alimentationChoisie.getAtom().allSupertypes()->includes(Alimentation)
```

Figure 6. *Contrainte relative à la définition d'entités alternatives*

3.1.4. *Entités avec contrainte d'exclusion mutuelle ou de dépendance*

Lorsque nous décrivons une ligne de produits, il nous faut aussi pouvoir spécifier que deux options ou deux entités ne peuvent coexister dans le même appareil. Pour cela, il faut ajouter dans la caractéristique ou le paramètre la contrainte adéquate. Par exemple si nous considérons à nouveau les figures 3 et 4 et que nous voulions spécifier qu'un appareil ne peut contenir à la fois un écran d'affichage (*affichage* de type *Écran*) et un port infra-rouge (*portIR* de type *PortInfraRouge*) mais seulement l'un ou l'autre (parce que les deux ensemble sont trop coûteux en énergie), nous ajouterons la contrainte proposée dans la figure 7. Pour exprimer une exclusion mutuelle entre deux entités alternatives, nous pourrions utiliser une approche similaire.

8. Comme vous avez pu le voir, le suffixe *Sem*, pour *sémantique*, identifie un concept par rapport à un atome, mais l'espace de nommage étant différent, il n'y a en fait aucune contrainte et le nom pourrait être identique.

```
context DefParOptionsChoisies
inv exclusionMutuelleEntreOptions : -- contrainte d'assemblage
let derivedAtom : Atom = concept.getCurrentAtom()
    derivedAtom.optionsChoisies->includes("portIR") xor
    derivedAtom.optionsChoisies->includes("affichage")
```

Figure 7. *Contrainte d'exclusion mutuelle entre des options*

De même, il est possible de décrire des dépendances entre des entités. Par exemple, la présence d'un port infrarouge (*PortIR*) dans un appareil nécessite une alimentation de type P2 (*AlimP2*). La figure 8 définit une contrainte de dépendance relative au contenu du paramètre *optionsChoisies* et de la caractéristique *AlimentationChoisie* dans *AppareilNumériqueSem*. Nous aurions aussi pu placer cette contrainte dans *DefParOptionsChoisies* ou dans *DefCharAlimentationChoisie* mais elle nous semble plutôt être de niveau méta du concept.

```
context AppareilPhotoSem
inv dependanceOptionsAlimentation : -- contrainte d'assemblage
let derivedAtom : Atom = getCurrentAtom()
    derivedAtom.optionsChoisies->includes("PortIR") implies
    derivedAtom.alimentationChoisie->includes("AlimP2")
```

Figure 8. *Contrainte de dépendance entre deux entités*

3.2. *Exprimer la variabilité du comportement*

Nous avons vu, dans la section 2, la place des *actions*. Le comportement qu'elles modélisent s'appuie à la fois sur la structure des atomes, sur les paramètres et sur les caractéristiques. Supposons que dans un appareil photographique nous modélisions le comportement lié à la communication avec l'extérieur comme le montre le modèle (figure 2) et plus particulièrement le fonctionnement de l'appareil pour ce point de vue. Une des propriétés de nos appareils est qu'ils communiquent de manière intelligente avec l'extérieur : c'est l'appareil qui choisit le port le mieux adapté. Ici nous avons retenu l'aspect « économie d'énergie » et donc le paramètre *OptimiserÉnergie* a été ajouté au concept (figure 9) de la même manière que les caractéristiques et paramètres l'ont été dans les sections précédentes.

```
nom : DefParOptimiserÉnergie -- définition d'un paramètre de comportement
rôle : mémoriser la faculté de limiter la consommation d'énergie
type : booléen
valeur par défaut : vrai
redéfinition possible dans un concept plus spécialisé : libre
```

Figure 9. *Définition d'un paramètre pour un appareil photographique*

L'action[9] qui est décrite dans la figure 10 recherche, à partir de la liste des objets de type *Port* connectés (c'est-à-dire les ports prêts à émettre ou recevoir des données), le port le plus économique en terme de consommation d'énergie. Pour cela elle se base sur les atomes présents dans le produit dérivé (un type d'appareil photographique particulier) et sur la valeur du paramètre *OptimiserÉnergie*.

La syntaxe utilisée est du pseudocode accompagné d'OCL ; elle pourra par la suite être adaptée pour se rapprocher de l'utilisateur. Nous pouvons y voir cinq parties : *i)* la signature de l'action (nom, paramètres et type de résultat), *ii)* une clause *required* qui indique les entités impliquées dans la définition de l'action, *iii)* une clause *require* décrivant ses préconditions, *iv)* le corps à proprement parler (à partir du *do*) et *v)* une clause *ensure* pour les postconditions.

```
SélectionnerPort(l : Collection<Port>) : Atom is
-- l = les ports qui sont actuellement en service
-- c'est-à-dire prêts à envoyer et/ou recevoir des données
required
    parameters : optionsChoisies, optimiserÉnergie
    characteristics : portsChoisis
    actions : none
require
    pre l->notEmpty and l->size >= 0
    pre getCurrentAtom().optionsChoisies->isUnique(true)
do
    let allPorts : set(Port) = Set(portUSB) -- USB obligatoire
    allPorts->union(optionsChoisies->select(e : EntityEnumeration |
    e.getAtom().supertypes->includes(Port)))
    -- allPorts = tous les ports assemblés de l'appareil
    if getCurrentAtom().OptimiserÉnergie and allPorts->includes(PortInfraRouge)
    and l->exists(e : Port | e.oclIsKindOf(PortInfraRouge)
    then
        result = PortInfraRouge -- suppose que c'est le plus économique
    else
        -- dépend des autres types de port possibles
    endif
ensure
    post result->notEmpty and result.allSupertypes->includes(Port)
end
```

Figure 10. *Action modélisant le comportement d'un appareil photographique*

Avant de passer à la mise en œuvre logicielle, nous concluons cette partie en résumant notre approche. Il s'agit donc de permettre la définition de métamodèles génériques qui permettent la génération, par instanciation et composition, de modèles métiers spécifiques aisément réutilisables.

4. Mise en œuvre

La mise en œuvre suit une approche dirigée par les modèles. C'est ainsi que nous avons étendu le modèle EMF d'Eclipse comme indiqué dans la section 2.1. En suivant

9. Cette action a été particulièrement simplifiée, notre but étant de donner un aperçu de l'expressivité.

la même philosophie que le modèle de base *Ecore* d'Eclipse qu'il étend, ce métamodèle est largement dédié à la saisie des informations relatives aux modèles métiers. D'ailleurs, notre éditeur a été généré à partir du métamodèle à l'aide des outils fournis par Eclipse et il a été ensuite retravaillé afin de le rendre plus simple d'utilisation. Cet éditeur permet de spécifier un modèle qu'il faut ensuite transformer afin d'aboutir à une bibliothèque de classes adaptées pour le développement d'applications dédiées. L'architecture de cette bibliothèque est décrite dans un deuxième modèle indépendant, appelé *built-in*. Elle réalise l'organisation en paquetages et classes de haut niveau pour définir les différentes hiérarchies, comme celles des atomes. À partir des deux modèles une première transformation va permettre d'obtenir un nouveau modèle EMF. Il représente le modèle de départ mais il est désormais, *i)* organisé suivant l'architecture définie dans le modèle *built-in*, *ii)* équipé de fonctionnalités dédiées notamment à faciliter l'accès aux concepts et à leur contenu, *iii)* complété par des informations représentées, pour le moment, sous forme de commentaires car non exprimables dans le modèle EMF. Comme le modèle généré est un modèle EMF, il peut être projeté vers le langage Java en utilisant le mécanisme de transformation proposé par Eclipse pour générer des classes java (*Java Emitter Templates - JET*). Naturellement, ce dernier a été modifié, en particulier pour prendre en compte les commentaires et compléter la génération des classes Java.

5. Travaux connexes

Beaucoup de travaux sur les lignes de produits s'appuient sur UML. En dehors de (Ziadi, 2004), il y a peu d'approches complètes qui incluent à la fois les parties statiques et dynamiques des modèles, ainsi que la dérivation des produits[10]. Par exemple, (Gomaa *et al.*, 2004) considère uniquement l'aspect statique du modèle, tandis que les travaux de (John *et al.*, 2002) sont basés sur les cas d'utilisation.

L'approche proposée par (Ziadi, 2004) consiste à *i)* utiliser un profil UML qui permet de décrire la variabilité, aussi bien dans les diagrammes statiques que dynamiques, *ii)* gérer des contraintes OCL pour exprimer à la fois des contraintes communes ou non à toutes les lignes de produits et *iii)* dériver, par transformation de modèles, les modèles statiques et dynamiques des lignes de produits. Dans cette approche, les mécanismes d'extension d'UML (stéréotypes, *tagged values*...) sont largement utilisés et la description de l'aspect dynamique du modèle est réalisée par des diagrammes de séquence (interactions entre les différentes entités du modèle) et par les machines à états pour la modélisation du comportement interne à un objet. Même si cette approche s'appuie, comme la nôtre, sur la transformation de modèles pour aboutir à un modèle exécutable, nous pouvons noter un certain nombre de différences. La première concerne la modélisation du comportement : T. Ziadi s'appuie sur le standard UML et ses différents diagrammes. Nous privilégions une approche plus programmatique basée sur la description d'actions encapsulées dans la partie méta des entités à modéliser et sur un mécanisme de composition de modèles qui seront ensuite assemblés en

10. (Ziadi, 2004) donne un état de l'art complet du domaine.

fonction des besoins. Ce mécanisme de composition permet de maintenir des actions de petite taille et de réduire la complexité de la description. Une seconde différence est que, dans notre approche, les produits dérivés sont construits par instanciation de la partie générique de la ou des entités associées. Dans son approche, T. Ziadi les construit en s'appuyant sur le patron de conception *fabrique abstraite* (Gamma *et al.*, 1999) et sur MTL, un langage de transformations de modèles (Projet Triskell, INRIA, 2005).

La mise en œuvre de la variabilité des diagrammes statiques et dynamiques, c'est-à-dire l'aspect optionnel ou alternatif de certaines entités (propriétés, méthodes, classes...), pourrait s'appuyer sur la séparation des préoccupations et plus particulièrement sur la programmation par aspects (Kiczales *et al.*, 2001) ou sur la programmation par sujets (Ossher *et al.*, 2000). Plusieurs travaux concernent la réalisation de lignes de produits logicielles (Lopez-Herrejon *et al.*, 2002) en utilisant AspectJ et Java. De ces études ressort que plus l'intégration des différents aspects s'appuie sur des hypothèses complexes, plus l'aide à l'intégration est limitée. Cela augmente significativement les risques d'erreur, par exemple la définition d'options incompatibles entre elles. Pour pallier cette absence de contrôle, il est nécessaire de disposer d'un protocole ou d'un langage de composition dédié à la variabilité des lignes de produits.

Comme nous l'avons vu dans la section 3, nous ne traitons pas la variabilité d'une ligne de produits en utilisant des approches basées sur la séparation des préoccupations mais sur la définition d'une généricité paramétrée. La variabilité pourrait être placée dans un modèle indépendant qui serait composé avec le modèle qui contient les entités de base de la ligne de produits au moment de l'instanciation de celle-ci. L'approche que nous avons définie dans (Lahire *et al.*, 2006; Crescenzo *et al.*, 2005) s'applique plus à la composition de modèles illustrant des fonctionnalités ou des points de vues différents qu'à une composition qui prend en compte la variabilité. Ces travaux sont voisins de de ceux de (Bouzitouna *et al.*, 2005) ou de (Muller *et al.*, 2003). Mais, pour leur appliquer notre approche, il faudrait modifier le langage dédié à la composition, qui est actuellement trop généraliste, afin qu'il réponde de manière plus ciblée aux besoins d'un concepteur de ligne de produits.

6. Conclusion et perspectives

Dans cet article, nous avons proposé une approche pour définir des modèles métiers qui s'appliquent plus particulièrement à des lignes de produits. Cette approche s'appuie sur la définition d'un degré de généricité pour chaque entité, en fonction des besoins, et sur la création des produits dérivés par simple instanciation des informations génériques. Nous nous démarquons des approches qui utilisent et étendent les diagrammes UML, en décrivant le comportement intrinsèque du modèle par des actions qui exploitent les paramètres et caractéristiques d'une entité générique. Nous définissons ainsi des applications dédiées en complétant des visiteurs générés et équipés spécifiquement pour le modèle associé.

Cette méthodologie implique que les modèles manipulés sont plutôt petits ce qui est le cas de beaucoup de modèles métiers. Nous défendons l'idée qu'en couplant notre approche avec un mécanisme de composition, nous devons obtenir une bonne réactivité et des développements rapides et évolutifs, Ce mécanisme permet de considérer indépendamment les différentes facettes d'un modèle pour ensuite les composer à partir de la description d'adaptateurs reposant sur un langage dédié.

Parmi les perspectives, nous continuons à affiner notre approche aussi bien pour la description du modèle métier que pour la composition de plusieurs modèles. Nous nous intéresserons en particulier à *i)* améliorer la qualité de la bibliothèque Java générée (lisibilité du code, nouvelles fonctionnalités...), *ii)* créer un langage dédié pour la définition des actions, *iii)* améliorer et compléter la mise en œuvre de la composition de modèles et *iv)* augmenter l'expressivité du support des contraintes OCL en proposant notamment de différencier les contraintes d'assemblage et d'exécution.

7. Bibliographie

Bouzitouna S., Gervais M.-P., Blanc X., « Models Reuse in MDA », *Proceedings of the 2005 International Conference on Software Engineering Research and Practice - SERP'05*, CSREA Press, Las Vegas, Nevada, USA, June, 2005. 8 pages.

Claub M., « Modeling Variability with UML », *Proceedings of Young workshop at GCSE'01*, Erfurt, Germany, p. 5, September, 2001.

Crescenzo P., OFL : un modèle pour paramétrer la sémantique opérationnelle des langages à objets - Application aux relations inter-classes, Thèse de doctorat, Université de Nice-Sophia Antipolis, France, décembre, 2001.

Crescenzo P., Lahire P., « Une approche pour améliorer la réutilisabilité des modèles métiers », *Actes de la 2ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*, Lille, p. 51-73, septembre, 2005.

Eclipse foundation, « Eclipse Environment », , http ://www.eclipse.org, 2004.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Catalogue de modèles de conception réutilisables*, Addison-Wesley Publishing Co., 1999.

Gomaa H., Webber D. L., « Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model », *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, Hawaii, USA, p. 268-277, January, 2004.

John I., Muthig D., « Tailoring Use Cases for Product Line Modeling », *Proceedings of the International Workshop on Requirements Engineering for Product Lines, REPL'02*, Essen, Germany, p. 26-32, September, 2002.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W., « An Overview of AspectJ », *in* , J. L. Knudsen (ed.), *Proceedings of ECOOP'01*, LNCS(2072), Springer Verlag, Budapest, Hungaria, June, 2001.

Lahire P., Quintian L., « New Perspective To Improve Reusability in Object-Oriented Languages », *Journal Of Object Technology (JOT)*, vol. 5, n° 1, p. 1-20, 2006.

Lopez-Herrejon R., Batory D., Using AspectJ to implement product-lines : a case-study, Technical report, University of Texas, 2002.

Muller A., Caron O., Carré B., Vanwormhoudt G., « Réutilisation d'aspects fonctionnels : des vues aux composants », *Actes de la conférence Langages, Modèles et Objets, LMO 2003*, Hermés Sciences, Vannes, France, p. 241-255, janvier, 2003.

OMG, *Unified Modeling Language (UML) OCL - Final Adopted Specification*, Object Management Group. October, 2003, Version 2.0 - undergoing finalization.

Ossher H., Tarr P., « Hyper/J : Multi-Dimentionnal Separation of Concern for Java », *in* , C. Ghezzy (ed.), *Proceedings of ICSE'00*, ACM Press, Limerick, Ireland, June, 2000.

Projet Triskell, INRIA, « MTL : Model Transformation Language », , http ://modelware.inria.fr/rubrique8.html, 2005.

Vanwormhoudt G., « Précision et Validation de Métamodèles avec EMF et OCL », *in* , P. Collet, , P. Lahire (eds), *Actes des journée Objects Composants et Modèles (OCM) du GDR ALP*, Berne, p. 19-28, Mars, 2005.

Ziadi T., Manipulation de lignes de produits en UML, Thèse de doctorat, Université de Rennes I, Rennes, France, décembre, 2004.